Go Code Generation for Isabelle

Terru Stübinger, Lars Hupel

April 18, 2024

Abstract

This entry contains a standalone code generation target for the Go programming language. Unlike the previous targets, Go is not a functional language and encourages code in an imperative style, thus many of the features of Isabelle's language (particularly data types, pattern matching, and type classes) have to be emulated using imperative language constructs in Go. To generate Go code, users can simply import this entry, which makes the Go target available.

```
theory Go-Setup
  imports Main
begin

⟨ML⟩

code-identifier
  code-module Code-Target-Nat ⇀ (Go) Arith
| code-module Code-Target-Int ⇀ (Go) Arith
| code-module Code-Numeral ⇀ (Go) Arith

code-printing
  constant Code.abort ⇀
    (Go) panic( - )


code-printing
  type-constructor bool ⇀ (Go) bool
| constant False::bool ⇀ (Go) false
| constant True::bool ⇀ (Go) true

code-printing
  constant HOL.Not ⇀ (Go) '! -
| constant HOL.conj ⇀ (Go) infixl 1 &&
| constant HOL.disj ⇀ (Go) infixl 0 ||
| constant HOL.implies ⇀ (Go) !('!((-)) || (-))
| constant HOL.equal :: bool ⇒ bool ⇒ bool ⇀ (Go) infix 4 ==




definition go-private-map-list where
  go-private-map-list f a = map f a
definition go-private-fold-list where
  go-private-fold-list f a b = fold f a b


code-printing
  type-constructor String.literal ⇀ (Go) string
| constant STR '''' ⇀ (Go)
| constant Groups.plus-class.plus :: String.literal ⇒ - ⇒ - ⇀
    (Go) infix 6 +
| constant HOL.equal :: String.literal ⇒ String.literal ⇒ bool ⇀
    (Go) infix 4 ==
| constant (≤) :: String.literal ⇒ String.literal ⇒ bool ⇀
    (Go) infix 4 <=
| constant (<) :: String.literal ⇒ String.literal ⇒ bool ⇀
    (Go) infix 4 <
```

⟨ML⟩

```
code-printing
  code-module Bigint ⇀ (Go) ‹
package Bigint

import math/big

type Int = big.Int;

func MkInt(s string) Int {
  var i Int;
  -, e := i.SetString(s, 10);
  if (e) {
    return i;
  } else {
    panic(invalid integer literal)
  }
}

func Uminus(a Int) Int {
  var b Int
  b.Neg(&a)
  return b
}

func Minus(a, b Int) Int {
  var c Int
  c.Sub(&a, &b)
  return c
}

func Plus(a, b Int) Int {
  var c Int
  c.Add(&a, &b)
  return c
}

func Times (a, b Int) Int {
  var c Int
  c.Mul(&a, &b)
  return c
}

func Divmod-abs(a, b Int) (Int, Int) {
  var div, mod Int
  div.DivMod(&a, &b, &mod)
  div.Abs(&div)
  return div, mod
}

func Equal(a, b Int) bool {
  return a.Cmp(&b) == 0
}

func Less-eq(a, b Int) bool {
  return a.Cmp(&b) != 1
}

func Less(a, b Int) bool {
  return a.Cmp(&b) == −1
}

func Abs(a Int) Int {
  var b Int
  b.Abs(&a)
  return b
}
› for constant uminus :: integer ⇒ - minus :: integer ⇒ - Code-Numeral.dup
Code-Numeral.sub
  (∗) :: integer ⇒ - (+) :: integer ⇒ - Code-Numeral.divmod-abs HOL.equal ::
integer ⇒ -
  less-eq :: integer ⇒ - less :: integer ⇒ - abs :: integer ⇒ -
  String.literal-of-asciis String.asciis-of-literal
  | type-constructor integer ⇀ (Go) Bigint.Int
  | constant uminus :: integer ⇒ integer ⇀ (Go) Bigint.Uminus( - )
  | constant minus :: integer ⇒ integer ⇒ integer ⇀ (Go) Bigint.Minus( -, -)
  | constant Code-Numeral.dup ⇀ (Go) !(Bigint.MkInt(2) ∗ -)
  | constant Code-Numeral.sub ⇀ (Go) panic(sub)
  | constant (+) :: integer ⇒ -  ⇀ (Go) Bigint.Plus( -, -)
  | constant (∗) :: integer ⇒ - ⇒ -  ⇀ (Go) Bigint.Times( -, -)
  | constant Code-Numeral.divmod-abs ⇀
     (Go) func () Prod[Bigint.Int, Bigint.Int] { a, b := Bigint.Divmod′-abs( -, -);
return Prod[Bigint.Int, Bigint.Int]{a, b}; }()
  | constant HOL.equal :: integer ⇒ - ⇀ (Go) Bigint.Equal( -, -)
  | constant less-eq :: integer ⇒ integer ⇒ bool  ⇀ (Go) Bigint.Less′-eq( -, -)
  | constant less :: integer ⇒ -  ⇀ (Go) Bigint.Less( -, -)
  | constant abs :: integer ⇒ - ⇀ (Go) Bigint.Abs( - )


code-printing
  constant 0::integer ⇀ (Go) Bigint.MkInt(0)
⟨ML⟩

end
```